Refactoring Code to Increase Readability and Maintainability

An Honors Thesis (HONR 499)

by

Christopher Dibble II

Thesis Advisor

Paul V. Gestwicki, Ph.D.

Ball State University

Muncie, Indiana

April 2013

Expected Date of Graduation

May 2013

SpColl
Undergrad
Thesis
LD
2489
.24
2013
.D52

Abstract

In this paper, I present a case study on the tool-mediated refactoring process and its effects, in terms of software metrics and performance, on an educational video game. The game is *Museum Assistant: Design an Exhibit*, which was released in May 2012 by an interdisciplinary team of undergraduate students. The game was developed for the Web in C# using Unity3D and MonoDevelop. The team set out to apply the principles of Clean Code throughout the development of the game. However, late in the development cycle, the team knowingly abandoned methodological constraints in the face of production stress. For this study, I began a six-week refactoring period comprising two phases: the first was mediated by the popular automated refactoring tool ReSharper, and the second involved manual identification of SRP violations. ReSharper's static analysis tools identified over 900 design problems, 95% of which were resolved in refactoring. In my analysis, I discuss how ReSharper simplified the detection and removal of superficial design defects such as naming convention violations, scope limitations, and idiomatic language syntax features, but it was not able to identify larger design flaws such as opportunities for design patterns. The details of one particular pattern-based refactoring are introduced in order to illustrate the relative merits of tool mediation and manual inspection: the introduction of a Chain of Responsibility. Code metrics, as computed by Visual Studio, were tracked throughout the refactoring process in order to estimate changes in readability and maintainability. I identify lines of code, readability index, class coupling, and cyclomatic complexity for extended discussion; all of these metrics improved through the refactoring process.

Disclaimer

This honors thesis is derived from C. Dibble, P. Gestwicki, and B. McNely *Tool-Mediated and Manual Refactoring of an Educational Video Game: A Case Study*, currently under review for publication. This work highlights the contributions of the first author.

Acknowledgements

I would first like to thank my advisor, Dr. Gestwicki, for helping me with this project. His encouragement throughout my college career has helped to build me into the person I am today. His guidance throughout this project was invaluable. I want to especially acknowledge his assistance in converting my work into the LaTeX format for both our co-authored paper and this thesis.

I would like to thank Dr. McNely for his contributions to the paper we co-authored midway through the semester.

I want to thank my parents, who have been there to encourage and support me throughout my entire life.

Finally, I would like to thank the members of Root Beer Float Studio for creating the game that I refactored during this project.

1. Introduction

After five months of design and development, Root Beer Float Studio released its first and only video game, *Museum Assistant: Design an Exhibit*, in late April 2012. The game places the player in the role of a museum volunteer who must visit the collection storage areas and cleverly assemble artifacts for an online exhibition. The team consisted of thirteen undergraduates from a variety of academic majors at Ball State University. The team worked together full-time during the Spring 2012 semester in an immersive learning context: that is, they worked under faculty supervision, for credit, on an artifact with a community partner [1, 6, 13]. Their game was designed in collaboration with The Children's Museum of Indianapolis to teach about the philosophy and operations of museums, including the spaces, jobs, processes, and constraints involved in collection, curation, and exhibit design [9].

The team had a firm deadline and faced significant pressure due to the necessity of design changes late in development [5]. Although the team had established a methodology based on industrial best practices—including test-driven development (TDD), pair programming, and rigorous coding conventions—they knowingly abandoned many of these in the face of the pressure to ship a game that would satisfy the community partner and player community. As a result, the team's code quality degraded as the deadline approached.

2. Refactoring Methodology

2.1 Background and Assumptions

The refactoring was mediated by the use of ReSharper, a popular add-on for Microsoft Visual Studio developed by JetBrains. The process was also mediated by my mentor, who had served as project mentor for Root Beer Float Studio and was available for consultation during refactoring. In practice, he was only consulted when I was uncertain about how to proceed given a ReSharper-reported issue or code smell [11].

The original development team was aware of its deviation from self-imposed methodological constraints. During the final retrospective of primary development, a majority of the development team lamented the fact that we had fallen away from the principles defined in Clean Code. As production stress increased and deadlines drew near, the team developed a laissez-faire attitude with respect for coding standards. This led to bloated, multipurpose methods that were in clear violation of the single responsibility principle. Given this perspective, we approached the refactoring process with expectations that the most serious defects would be SRP violations found in the most recently written code. It was expected that ReSharper would help identify defects that had existed longer, but that the density and complexity of defects would increase with time.

2.2 Tools and Techniques

In order to determine the effect of refactoring on the codebase, we decided to use Visual Studio's built-in code metrics to track changes in project structure and complexity over time. The built-in metrics include cyclomatic complexity [12], depth of inheritance, class coupling, lines of code, and maintainability index [14], which is a composite of Halstead volume [7], cyclomatic complexity, and lines of code. While we monitored the effect of the refactorings on each of the metrics, special attention was given to the maintainability index as a measure of how far the project had come.

The refactoring process was divided into two phases. The first phase involved using ReSharper to identify issues throughout the code and then addressing these issues using ReSharper's recommendations and autofix options. Automated refactorings were applied on each file as recommended by ReSharper. It took approximately 10–12 person-hours to process issues the 111 source files.

The second phase of the refactoring process consisted of a six week manual inspection. Following the initial sweep of the files, we revisited each file and performed refactoring based on the principles established by Fowler [3], Martin [11], and Wagner [16]. As mentioned earlier, Martin had been chosen by the team as part of its methodology; Fowler was appopriate for framing the refactoring process, and Wagner was chosen based on its perceived popularity in industry.

3. Refactoring Results and Discussion

3.1 Metrics

At the conclusion of the refactoring process we had many interesting results in both the benchmark changes and the way in which the number of issues identified by ReSharper diminished over time. Benchmark changes are described in Table 1. The table shows that we saw no significant negative impact on the code metrics after refactoring. Four of the five metrics improved, with one remaining unchanged. We find that the refactored code is more readable and understandable, particularly subject to the principles articulated by Fowler [3], Martin [11], and Wagner [16]; this subjective opinion is reflected quantitatively in the metrics. Note that depth of inheritance is measured here only over the source code developed by Root Beer Float Studio. Integration with Unity3D requires subclassing an abstract class, `MonoBehaviour`. Counting this class' superclasses increases the depth of inheritance by three; however, since the class is a black box, we chose to treat it as a single level regardless of what inaccessible superclasses it may have.

Although the changes we were able to achieve are substantial, we believe they are muted because of the initial overall high quality of the code base. Microsoft's guidelines suggest that any maintainability above 20 is considered to be maintainable [14]. An upper bound for cyclomatic complexity is ten [17], and for class coupling, nine [15]. Based

Metric	Pre-refactor	Post-refactor	% Change
Maintainability Index	85	87	+2.35
Average Cyclomatic Complexity	1.41	1.30	-7.91
Depth of Inheritance	2	2	0
Average Class Coupling	2.53	2.27	-10.40
Lines of Code	2823	2669	-5.46
Number of files	111	112	+0.89

Table 1. Metrics before and after refactoring

on the recommended values for cyclomatic complexity, class coupling, and maintainability index, the project was in fine shape from the onset. However, we were still able to achieve significant reduction and primarily through tool-mediated refactoring via ReSharper.

One significant fact to note is that we saw a 5.46% reduction in the total lines of code while still seeing improvement in the other metrics. This is significant because one might expect the lines of code to increase as additional classes, methods, and other abstractions are added. In our case, we were able to achieve dramatic reductions in the cyclomatic complexity and class coupling while still reducing the overall lines of code.

The C# compiler will generate constructors where they are not specified. These default constructors are included in the Visual Studio metrics computation, and each has a maintainability index of 100, perfect class complexity, and a coupling rating of zero: including these in the metrics inflates the composite scores. The values that we report are based only on the code we wrote, with the metrics for these rows manually eliminated. This behavior is documented on the MSDN Web site, but it is not obvious from the presentation of the data. It is also not clear that metrics for default constructors are useful when one is interested in source code generated by software developers.

3.2 C# conventions

During the original development process, few team members had any significant C# development experience. As a result, the naming practices used in the project were a hybrid of official Java standards and Microsoft-recommended C# standards. These conventions were not consistent between or even within files. ReSharper was able to quickly point out the locations where the team had diverged from the C# conventions. It was also able to automatically refactor such offending code, dramatically increasing readability with almost no human intervention required.

ReSharper guided the refactoring process through several other changes that leverage syntactic features of C#. It identified many instances where the scope of a method or variable could safely be more restricted, resulting in higher-quality code. Many class and instance variables were made `readonly` where, in practice, they were being used in a manner consistent with the keyword. Where objects had been configured in multiple lines of code, object initializers were introduced, with properties specified immediately after instantiation. For example, the following two lines existed in the original code.

```
var map = new MapIconButton();
map.Enabled = true;
```

They were changed to the following:

```
var map = new MapIconButton {Enabled=true};
```

3.3 LINQ

ReSharper guided the refactored code to make use of Language-Integrated Query (LINQ), a language feature that allows the insertion of SQL-like queries directly into C# code [2]. With LINQ statements, it is sometimes possible to convert entire loop structures into a single, simple, LINQ expression. As an example, consider the following (pre-refactored) code from the domain model:

```
public bool ContainsThreeOfATheme()
{
  foreach (Object theme in metaDataSet.Keys)
  {
    if (_metaDataSet[theme]==3)
    {
      return true;
    }
  }
  return false;
}
```

Following ReSharper's recommendations, we were able to use LINQ to convert the for-each loop into a single return statement:

```
public bool ContainsThreeOfATheme()
{
  return _metaDataSet.Keys.Any
     (theme => _metaDataSet[theme]==3);
}
```

In many cases we were able to replace multiline loops with easy to understand LINQ expressions. However, we found that each recommended change required careful consideration since, in some cases, the LINQ statements were no more readable than the original code—and sometimes much less so. For example, for the original implementation in Listing 1, ReSharper recommended the replacement shown in Listing 2; we argue that the "refactored" is more difficult to comprehend than the original version.

Listing 1. LINQ Example—Original implementation

```
foreach (TextAsset asset in Resources.LoadAll("Rooms/XML",typeof(TextAsset))
{
  RoomContainer room = xmls.Deserialize(new StringReader(asset.text)) as RoomContainer;
  roomContainers.Add(room);
}
```

Listing 2. LINQ Example—Recommended refactoring

```
return (from TextAsset asset in Resources.LoadAll("Rooms/XML", typeof(TextAsset))
        select xmls.Deserialize(new StringReader(asset.text)) as RoomContainer).toList();
```

Category	Pre-refactor	Post-refactor	% Reduction
Common Practices and Code Improvements	178	0	100.00
Compiler Warnings	1	0	100.00
Constraints Violations	245	0	100.00
Language Usage Opportunities	130	16	87.69
Potential Code Quality Issues	34	10	70.59
Redundancies in Code	304	0	100.00
Redundancies in Symbol Declarations	78	27	65.38
Total Issues	970	53	94.54

Table 2. Defects identified by ReSharper and before and after refactoring.

3.4 ReSharper Issues

ReSharper has an "Inspect" feature that will analyze code projects for problems or refactoring suggestions. During this project, we utilized this feature to both identify common problems within the code and to track the progress with which these issues were resolved. At the onset of our project, ReSharper identified 970 potential issues within the code. By the end of the refactoring process, that number had decreased to 53. Table 2 describes the change in number of issues broken down into categories. The 53 remaining issues reported by ReSharper at the end of the project are a result of conscious decisions on our part to disregard certain ReSharper recommendations. Table 3 details the specific types of issues that remained, as well as the reasons we had for disregarding their existence.

Figure 1 shows the total number of issues identified by ReSharper during the refactoring process. The figure illustrates how a large number of issues were able to be addressed in a very short amount of time by introducing the tool and leveraging it during refactoring. Note that ReSharper was deployed primarily in the first stage of refactoring, which was the first week of effort; the remaining reductions are from manual inspection.

While the first phase of refactoring was able to dramatically reduce the number of issues identified by ReSharper, it was also the only period during which the code metrics became worse. Compare Figure 1 to Figures 2 and 3, which track the changes in class coupling and cyclomatic complexity over time. One can clearly see that during phase one of

Figure 2. Average coupling over time

Figure 3. Average cyclomatic complexity over time

# Remaining	Issue Text	Reason for disregard
27	Method ⟨*f*⟩ is never used.	These methods were called from the Unity3D engine; ReSharper was not able to detect their usage.
13	Use implicitly typed local variable declaration	In these cases, we felt the type of the variable was not obvious enough to support implicit typing.
6	Possible System.NullReferenceException	These were false-positives that, due to the Unity3D API design, would not result in null references in practice.
3	For-loop can be converted into foreach-loop	The loops contained code that modified the collection, and so such a conversion would result in an invalid operation exception at runtime.
2	Possible unassigned object created by 'new' expression.	These were instantiations for which no persistent reference was required.
1	Impure method is called for readonly field of value type.	Calling `Contains` on a readonly rectangle, which does not modify state and is safe.
1	Similar expressions comparison.	False positive on a unit test for identity equality via `Equals`.

Table 3. Disregarded ReSharper issues and rationale

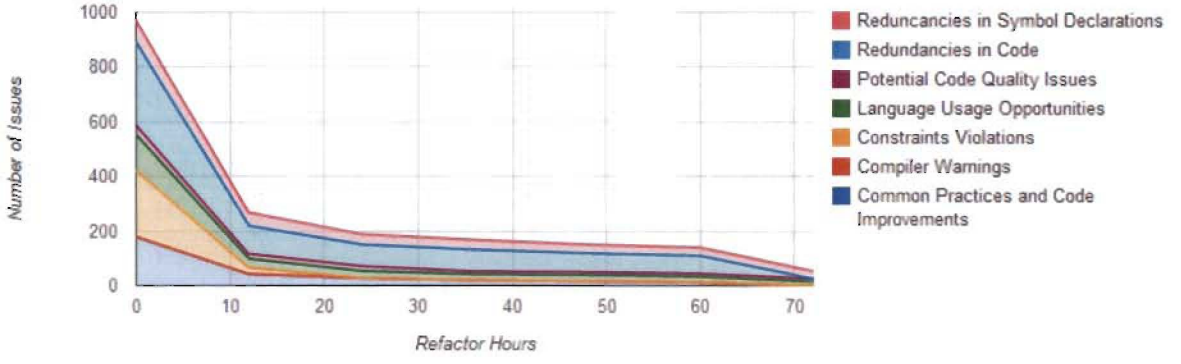

Figure 1. ReSharper Issue Timeline

refactoring, while ReSharper issues plummeted, both coupling and cyclomatic complexity increased. During the second phase of refactoring—manual inspection of all project files, with on-demand expert consultation—ReSharper issues remained low while coupling and cyclomatic complexity both improved dramatically.

Phase two of the refactoring process was focused on identifying and removing violations of the Single Responsibility Principle [10]. ReSharper was largely unable to assist during this process. On occasion, it was able to find possible ways to reduce nesting, but often these were done by inverting an if statement or introducing a LINQ expression in a manner that did not address deeper design issues. For example, ReSharper was unable to offer any suggestions on how to improve the structure of the method in Listing 3, which we describe in detail in Section 3.6. In the absence of useful mediation from ReSharper or Visual Studio, identifying and fixing these kinds of problems was an entirely manual operation. Knowledge of design patterns and code smells remained the best method for finding these problems within the system.

3.5 Runtime performance

Through the refactoring process, we made hundreds of modifications and touched every source code file in the system. We measured the performance of the system before and af-

ter refactoring using Unity3D's built-in profiler. Using the profiler introduces some overhead, but this impact should be agnostic of the code design. This same profiler was used during the game's original development to identify and remove bottlenecks, which were most frequently from unnecessary creation of objects during per-frame updates. We note that this FPS measurement is not a perfect measure of game performance, in part because actual FPS is limited by the display device; however, it is a standard measurement in game development, and provides an indicator that our game's performance was significantly improved.

Museum Assistant does not support gameplay scripting, and so we ran each of our tests ten times—five on the original release and five on the refactored code. First, we measured idle FPS on the title screen, where the original release achieved 185 average FPS and the refactored version, 446. To see if these results carried into gameplay, we stepped through the game's tutorial and first challenge with the profiler running. In this case, the original release achieved 370 average FPS and the refactored version, 366. From these values we conclude that refactoring must have removed unnecessary code from the title screen, but that taken as a whole, the refactoring did not have significant impact on the game's performance.

3.6 Chain of Responsibility

One notable refactoring during the second phase was the replacing of a series of conditionals with a chain of responsibility [4]. This example illustrates a refactoring to patterns as described by Kerievsky [8]. This refactoring is notable for our case study because, at this point in automated tool design, it requires human insight and creativity: ReSharper and VisualStudio were unable to provide guidance on this refactoring, but it provided a significant improvement to the maintainability and readability of the implementation.

The relevant software feature for this refactoring is the generation of appropriate user feedback upon failure to satisfy a curator's criteria for an exhibit. Listing 3 shows the original implementation, slightly edited for publication purposes. A consultationwith my mentor revealed the opportunity for the chain of responsibility, and the refactored method is shown in Listing 4. The chain is initialized as shown in Listing 5, with a sample handler implementation in Listing 6. Table 4 shows the impact of this refactoring on the metrics for this module, namely, an improvement in all except lines of code.

Listing 5 also illustrates how the builder pattern was used in collaboration with the chain of responsibility. The builder pattern facilitates writing functions with few parameters, in keeping with Clean Code standards [11], and it was used extensively in the domain model of the original implementation. During refactoring, we found that introducing this idiom in new places enhanced both readability and maintainability.

Metric	Before	After
Average maintainability index	61.0	92.94
Average Cyclomatic complexity	7.0	1.59
Class coupling	3.0	2.0
Lines of code	11	20

Table 4. Module metrics before and after the chain of command refactoring

4. Conclusions

Through this case study, we were able to show the effect of the automated refactoring tool ReSharper versus manual refactoring. We were also able to show that runtime performance remained constant despite the fact that we made hundreds of improvements to the readability and maintainability of the project.

The first phase demonstrated how, using ReSharper, a single developer was able to rapidly address hundreds of software design problems. These included naming convention violations, unnecessary code, and missed opportunities for advanced language features. However, ReSharper was only useful for identifying superficial changes—not changes that required deeper structural alterations. To get at these, a second manual inspection of each source file, focused on identifying and removing SRP violations via design patterns, was required. That significant improvement in the code metrics was observed as part of phase two and not phase one lends support to the assertion that the ReSharper mediated changes were largely superficial.

The improvements in code metrics show that the maintainability of the code base has improved significantly since the onset of this project. Any future maintainers of the game will be able to more quickly and effectively acclimate themselves to the project. The efforts we made to improve the maintainability of this game simplifies the future maintenance without sacrificing player experience.

Listing 3. Unrefactored Implementation of Failure Feedback Generation

```
private static string GetExhibitChallengeFailureFeedback (Tableau tableau, Challenge challenge)
{
  if (challenge.ForbidsPre1800() && tableau.ContainsMoreThan1Pre1800sEra()) {
    return "You shouldn't use Pre-1800 Era artifacts in this exhibit... You should try again.";
  }
  if (HasBlankPicture (tableau)) {
    return "One of these pictures doesn't even show an artifact... You should try again.";
  }
  if (ArtifactsHaveNothingInCommon (tableau)) {
    return "Oh no. These artifacts have nothing in common... You should try again.";
  }
  if (ArtifactsHaveNoThreeThemesAndOnlyTwoOfOneThemeInCommon(tableau)) {
    return "All the artifacts need at least one theme in common... You should try again.";
  }
  if (ArtifactsHaveTwoOfOneThemeInCommon(tableau)) {
    return "Close, but one of the artifacts seems out of place... You should try again.";
  }
  throw new Exception("Should never get here.");
}
```

Listing 4. Refactored Implementation of Failure Feedback

```
private static string GetExhibitChallengeFailureFeedback (Tableau tableau, Challenge challenge)
{
  return new ChallengeFailedFeedbackGenerator().GenerateFeedbackText(tableau, challenge);
}
```

Listing 5. Partial Implementation of `ChallengeFailedFeedbackGenerator`

```
_feedbackHandlerChain = new FailureFeedbackGeneratorBuilder()
  .FollowedBy(new IllegalPre1800Handler())
  .FollowedBy(new BlankPictureHandler())
  .FollowedBy(new NothingInCommonHandler())
  .FollowedBy(new NoCommonThemeHandler())
  .FollowedBy(new OneArtifactOutOfPlaceHandler())
  .FollowedBy(new LastResortHandler())
  .Build();
```

Listing 6. Sample Chain of Command Handler Implementation

```
class BlankPictureHandler : ChainOfCommandMember
{
  public override string HandleRequest(Tableau tableau, Challenge challenge)
  {
    return HasBlankPicture(tableau)
           ? NoArtifactInPictureMessage
           : Successor.HandleRequest(tableau, challengeToOvercome);
  }

  private bool HasBlankPicture(Tableau tableau)
  {
    return tableau.ContainsBlankMetaData();
  }
}
```

References

[1] J. Blackmer. The gesture of thinking: Collaborative models for undergraduate research in the arts and humanities—plenary presentation at the 2008 CUR national conference. *CUR Quarterly*, 29(1):8–12, 2008.

[2] D. Box and A. Hejlsberg. The LINQ Project: .NET Language Integrated Query. Microsoft Whitepaper, 2005.

[3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, 1999.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1994.

[5] P. Gestwicki and B. McNely. A case study of a five-step design thinking process in educational museum game design. In *Proceedings of Meaningful Play*, 2012.

[6] P. Gestwicki and R. Morris. Social studies education game development as an undergraduate immersive learning experience. In M. M. Cruz-Cunha, editor, *Serious Games as Educational, Business, and Research Tools: Development and Design*, pages 838–858. IGI Global, 2012.

[7] M. H. Halstead. *Elements of Software Science*. Elsevier, Holland, 1977.

[8] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, Boston, MA, 2004.

[9] B. Lord and G. D. Lord, editors. *The Manual of Museum Exhibitions*. Altamira Press, Lanham, MD, 2001.

[10] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, NJ, 2002.

[11] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, Upper Saddle River, NJ, 2008.

[12] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[13] J. McKillip. Transformative undergraduate research: Students as the authors of and authorities on their own education. *CUR Quarterly*, 30(2):10–15, 2009.

[14] MSDN Blog. Maintainability index range and meaning. `http://blogs.msdn.com/b/codeanalysis/archive/2007/11/20/maintainability-index-range-and-meaning.aspx`, 2007.

[15] R. Shatnawi. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on Software Engineering*, 36(2):216–225, 2010.

[16] B. Wagner. *Effective C#: 50 Specific Ways to Improve your C#*. Addison-Wesley, Boston, MA, second edition, 2010.

[17] A. H. Watson and T. J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-235, 1996.